High Performance Direct Gravitational N-body Simulations on Graphics Processing Units

Simon F. Portegies Zwart [a,b] Robert G. Belleman [a]
Peter M. Geldof [a]

[a] *Section Computational Science, University of Amsterdam, Amsterdam, The Netherlands*

[b] *Astronomical Institute "Anton Pannekoek" , University of Amsterdam, Amsterdam, The Netherlands*

Abstract

We present the results of gravitational direct N-body simulations using the commercial graphics processing units (GPU) NVIDIA Quadro FX1400 and GeForce 8800GTX, and compare the results with GRAPE-6Af special purpose hardware. The force evaluation of the N-body problem was implemented in Cg using the GPU directly to speed-up the calculations. The integration of the equations of motions were, running on the host computer, implemented in C using the 4th order predictor-corrector Hermite integrator with block time steps.

We find that for a large number of particles ($N \gtrsim 10^4$) modern graphics processing units offer an attractive low cost alternative to GRAPE special purpose hardware. A modern GPU continues to give a relatively flat scaling with the number of particles, comparable to that of the GRAPE. Using the same time step criterion the total energy of the N-body system was conserved better than to one in 10^6 on the GPU, which is only about an order of magnitude worse than obtained with GRAPE. For $N \gtrsim 10^6$ the GeForce 8800GTX was about 20 times faster than the host computer. Though still about an order of magnitude slower than GRAPE, modern GPU's outperform GRAPE in their low cost, long mean time between failure and the much larger onboard memory; the GRAPE-6Af holds at most 256k particles whereas the GeForce 8800GTF can hold 9 million particles in memory.

Key words: gravitation – stellar dynamics – methods: N-body simulation – methods: numerical –

1 Introduction

Since the first large scale simulations of self gravitating systems the direct N-body method has gained a solid footing in the research community. At the moment N-body techniques are used in astronomical studies of planetary systems, debris discs, stellar clusters, galaxies all the way to simulations of the entire universe (Hut, 2007). Outside astronomy the main areas of research which utilise the same techniques are molecular dynamics, elementary particle scattering simulations, plate tectonics, traffic simulations and chemical reaction network studies. In the latter non-astronomical applications, the main force evaluating routine is not as severe as in the gravitational N-body simulations, but the backbone simulation environments are not very different.

The main difficulty in simulating self gravitating systems is the lack of anti-gravity, which results in the requirement of global communication; each object feels the gravitational attraction of any other object.

The first astronomical simulation of a self gravitating N-body system was carried out by Holmberg (1941) with the use of 37 light bulbs and photoelectric cells to evaluate the forces on the individual objects. Holmberg spent weeks in order to perform this quite moderate 37-particle simulation. Over the last 60 or so years many different techniques have been introduced to speed up the kernel calculation. Today, such a calculation requires about 50 000 integration steps for one dynamical time unit. At a speed of $\sim 10\,\mathrm{Gflop/s}$ the calculation would be performed in a few seconds.

The gravitational N-body problem has made enormous advances in the last decade due to algorithmic design. The introduction of digital computers in the arena (von Hoerner, 1963; Aarseth & Hoyle, 1964; van Albada, 1968) led to a relatively quick evaluation of mutual particle forces. Advanced integration techniques introduced to turn the particle forces in a predicted space-time trajectory, opened the way to predictable theoretical results (Aarseth & Lecar, 1975; Aarseth, 1999). One of the major developments in the speed-up and improved accuracy of the direct N-body problem was the introduction of the block-time step algorithm (Makino, 1991; McMillan & Aarseth, 1993).

In the late 1980s it became quite clear that the advances of modern computer technology via Moore's law (Moore, 1965) was insufficient to simulate large star clusters by the new decade (Makino & Hut, 1988, 1990). This realization brought forward the initiatives employed around the development of special hardware for evaluating the forces between the particles (Applegate et al., 1986; Taiji et al., 1996; Makino & Taiji, 1998; Makino, 2001; Makino et al., 2003), and of the efficient use of assembler code on general purpose hardware (Nitadori, Makino, & Hut, 2006; Nitadori, Makino, & Abe, 2007).

One method to improve performance is by parallelising force evaluation Eq. 1 for use on a Beowulf or cluster computer (with or without dedicated hardware)(Harfst et al., 2006), a large parallel supercomputer (Makino, 2002; Dorband et al., 2003) or for grid operations (Gualandris et al., 2007). In particular for distributed hardware it is crucial to implement an algorithm that limits communication as much as possible, otherwise the bottleneck simply shifts from the force evaluation to interprocessor communication.

A breakthrough in direct-summation N-body simulations came in the late 1990s with the development of the GRAPE series of special-purpose computers (Makino & Taiji, 1998), which achieve spectacular speedups by implementing the entire force calculation in hardware and placing many force pipelines on a single chip. The latest special purpose computer for gravitational N-body simulations, GRAPE-6, performs at a peak speed of about 64 Tflop/s (Makino, 2001).

In our standard setup, one GRAPE-6Af processor board is attached to a host workstation, in much the same way that a floating-point or graphics accelerator card is used. We use a smaller version: the GRAPE-6Af which has four chips connected to a personal workstation via the PCI bus delivering a theoretical peak performance of $\sim$ 131 Gflops for systems of up to 128k particles at a cost of $\sim$ \$6K (Fukushige et al., 2005). Advancement of particle positions $[\mathcal{O}(N)]$ is carried out on the host computer, while interparticle forces $[\mathcal{O}(N^2)]$ are computed on the GRAPE.

The latest developments in this endeavour is the design and construction of the GRAPE-DR, the special purpose computer which will break the Pflop/s barrier by the summer of 2008 (Makino, 2007) [1] . One of the main arguments to develop such a high powered and relatively diverse computer is to perform simulations of entire galaxies (Makino, 2005a; Hoekstra et al, 2007).

The main disadvantages of these special purpose computers, however, are the relatively short mean time between failure, the limited availability, the limited applicability, the limited on-board memory to store particles, the simple fact that they are basically build by a single research team led by prof. J. Makino and the lack of competing architectures.

The gaming industry, though not deliberately supportive of scientific research, has been developing high power parallel vector processors for performing specific rendering applications, which are in particular suitable for boosting the frame-rate of games. Over the last 7 years graphics processing units (GPUs) have evolved from fixed function hardware for the support of primitive graphical operations to programmable processors that outperform conventional CPUs, in particular for vectorizable parallel operations. Regretfully,

[1] See `http://grape.astron.s.u-tokyo.ac.jp/grape/computer/grape-dr.html`

the precision of these processors is still 32-bit IEEE which is below the average general purpose processor, but for many applications it turns out that the higher (double) precision is not crucial or can be emulated at some cost. It is because of these developments, that more and more people use the GPU for wider purposes than just for graphics (Fernando, 2004; Pharr & Fernando, 2005; Buck et al., 2004). This type of programming is also called general purpose computing on graphics processing units (GPGPU) [2] . Earlier attempts to use a GPU for gravitational N-body simulations were carried out approximate force evaluation methods using shared time steps Nyland et al (2004), but provide little improvement in performance. A 25-fould speed increase compared to an Intel Pentium IV processor was reported by Elsen et al. (2006), but details of their implemtation of the force evaluation alrgorithm are yet unclear.

Using the GPU as a general purpose vector processor works as follows. Colours in a computer are represented by one or more numbers. The luminance can be represented by just a single number, whereas a coloured pixel may contain separate values indicating the amount of red, green and blue. A fourth value alpha may be included to indicate the amount of transparency. Using this information, a pixel may be drawn. There are many pixels in a frame, and ideally, these should be updated all at the same time and at a rate exceeding the response time of the human eye. This requires fast computations for updating the pixels, for example when a camera moves or a new object comes into view. Such operations usually have an impact on many or even all pixels fast computations are required. But since the majority of pixes do not require information from other pixes, processing can be done efficiently in parallel. All information required to build a pixel should go through a series of similar operations, a technique which is better known as single instruction, multiple data (SIMD). There are many different kinds of operations this information needs to go through. The stream programming model has been designed to make the information go through these operations efficiently, while exposing as much parallelism as possible. The stream programming model views all informations as "streams" of ordered data of the same data type. The streams pass through "kernels" that operate on the streams and produce one or more streams as output.

In this paper we report on our endeavour to convert a high precision production quality N-body code to operate with graphics processor units. In § 2 we explain the adopted N-body integration algorithm, in § 3 we address the programming environment we used to program the GPU, In the sections § 4 and § 5 we present the results on two GPUs and compare them with GRAPE-6Af and we discuss a model to explain the GPUs performance. In § 6 we summarise our findings, and in the Appendix we present a snippet of the source code in

[2] see `http://www.gpgpu.org`

Cg.

2 Calculating the force and integrating the particles

The gravitational evolution of a system consisting of N stars with masses m_j and at position $\mathbf{r}_j$ is computed by the direct summation of the Newtonian force between each of the N stars. The force $\mathbf{F}_i$ acting on particle i is then obtained by summation of all other $N-1$ particles

$$\mathbf{F}_i \equiv m_i \mathbf{a}_i = m_i G \sum_{j=1, j\neq i}^{N} m_j \frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|^3}. \tag{1}$$

Here G is the Newton constant.

A cluster consisting of N stars evolves dynamically due to the mutual gravity of the individual stars. For an accurate force calculation on each star a total of $\frac{1}{2}N(N-1)$ partial forces have to be computed. This $\mathrm{O}(N^2)$ operation is the bottleneck for the gravitational N-body problem.

The GPU scheme described in this paper is implemented in the N-body integrator. Here particle motion is calculated using a fourth-order, individual-time step "Hermite" predictor-corrector scheme (Makino and Aarseth 1992). This scheme works as follows. During a time step the positions ($\mathbf{x}$) and velocities ($\mathbf{v} \equiv \dot{\mathbf{x}}$) are first predicted to fourth order using the acceleration ($\mathbf{a} \equiv \ddot{\mathbf{x}}$) and the "jerk" ($\mathbf{k} \equiv \dot{\mathbf{a}}$, the time derivative of the acceleration) which are known from the previous step.

The predicted position ($\mathbf{x}_p$) and velocity ($\mathbf{v}_p$) are

$$\mathbf{x}_p = \mathbf{x} + (\mathbf{v} + (dt/2)(\mathbf{a} + (dt/3)\mathbf{k}))dt, \tag{2}$$
$$\mathbf{v}_p = \mathbf{v} + (\mathbf{a} + (dt/2)\mathbf{k})dt. \tag{3}$$

The acceleration and jerk are then recalculated at the predicted time, using x_p and v_p. Finally, a correction is based on the estimated higher-order derivatives:

$$\mathbf{a3} = 2(\mathbf{a} - \mathbf{a}_p) + (\mathbf{k} + \mathbf{k}_p)dt, \tag{4}$$
$$\mathbf{a2} = -3(\mathbf{a} - \mathbf{a}_p) - (2\mathbf{k} + \mathbf{k}_p)dt. \tag{5}$$

where

$$\mathbf{a2} = \dot{\mathbf{k}} dt^2/2, \tag{6}$$
$$\mathbf{a3} = \ddot{\mathbf{k}} dt^3/6. \tag{7}$$

Which then leads to the new position and velocity at time $t + dt$.

$$\mathbf{x} = \mathbf{x}_p + (\mathbf{a2}/12 + \mathbf{a3}/20)dt^2, \tag{8}$$
$$\mathbf{v} = \mathbf{v}_p + (\mathbf{a2}/3 + \mathbf{a3}/4)dt. \tag{9}$$

The new $\mathbf{a}$ and $\mathbf{k}$ are computed by direct summation, and the motion is subsequently corrected using the additional derivative information thereby obtained.

A single integration step in the integrator proceeds as follows:

- Determine which stars are to be updated. Each star has an individual time (t_i) associated with it at which it was last advanced, and an individual time step (dt_i). The list of stars to be integrated consists of those with the smallest $t_i + dt_i$. Time steps are constrained to be powers of 2, allowing "blocks" of many stars to be advanced simultaneously (McMillan & Aarseth, 1993).
- Before the step is taken, check for system reinitialization, diagnostic output, termination of the run, storing data.
- Perform low-order prediction of all particles to the new time $t_i + dt_i$. This operation may be performed on the GPU, if available.
- Recompute the acceleration and jerk on all stars in the current block (using the GPU, if available), and correct their positions and velocities to fourth-order.

Note that this scheme is rather simple as it does not include treatment for close encounters, binaries or higher order (hierarchical or democratic) stable multiple systems.

3 The programming environment

The part of the algorithm that executes on the GPU (the force evaluation) is implemented in the Cg computer language (C for graphics, Fernando & Kilgard (2003), see Appendix A), which has a syntax quite similar to C. The Cg programming environment includes a compiler and run-time libraries for use with the open graphics library (OpenGL) [3] and DirectX [4] graphics application programming interfaces. Though originally developed for the creation of real-time

[3] see `http://www.opengl.org`
[4] see `http://www.microsoft.com/directx`

Table 1
Detailed information on the hardware used in our experiments. The first column gives the parameter followed by the four different hardware setups (GRAPE-6Af, GeForce 8800GTX, Quadro FX1400 and information about the host computer. The information for the GRAPE is taken from Makino et al. (2003), the GPU information is from `http://www.nvidia.com`. The hardware details are the number of processors pipelines ($n_{\rm pipe}$), the processor's clock frequency ($f_{\rm GPU} \equiv 1/t_{\rm GPU}$), the memory bandwidth for communication between host and attached processor ($1/t_{\rm bus}$), the amount of memory (in number of particles, one particle requires 84 bytes, here we adopt $1{\rm k} \equiv 1024$). For measured hardware parameters, see Tab. 3.

data	GRAPE-6Af	8800GTX	FX 1400	Xeon	unit
$n_{\rm pipe}$	48	128	12	1	
$f_{\rm GPU}$	90	575	350	3400	MHz
$1/t_{\rm bus}$	33.8	86.4	19.2	NA	GB/s
Memory	128k	9362k	1562k	–	particles

special effects without the need to program directly to the graphics hardware assembly language, researchers soon recognised the potential of Cg and started to apply it not only to high-performance graphics but also to a wide variety of "general-purpose computing" problems (Fernando, 2004; Pharr & Fernando, 2005).

3.1 Mapping the N-body problem to a GPU

The challenge in the implementation of an efficient N-body code on a GPU lies in the mapping of the algorithm and the data to graphical entities supported by the Cg language. Particle data arrays are represented as "textures". Normally, textures are used to represent pixels colour attributes with one single component (luminance, red, green, blue or alpha), three components (red, green and blue) or four components (red, green, blue and alpha). In our implementation we use multiple textures to represent the input and output data of N particles, as follows:

- Input: mass (N), position ($3N$) and velocity ($3N$)
- Output: acceleration ($3N$), jerk ($3N$) and potential (N)

All values are represented as single precision (32-bit) floating point numbers for a total of 21 floats or 84 bytes per particle. In Appendix A we present a snippet of the source code in Cg, showing the implemented force evaluation routine. With the 768 Mbyte on-board memory of the GeForce 8800GTX it can store about 9 million particles, whereas the GRAPE-6Af can store only 128k (see Tab. 1).

Transferring data from CPU to GPU is accomplished through the definition of textures, which can either be read-only or write-only, but not both at the same time. The data structures in the CPU are then copied onto appropriately defined textures in the graphics card's memory. Obtaining the results from GPU to CPU is done by reading back the pixels from the appropriate rendering targets into data structures on the host CPU. Therefore the output textures (acceleration, jerk and potential) are represented by a double-buffered scheme, where after each GPU computation the textures are swapped between reading and writing. There is some additional overhead (of order N) for this operation which has to be performed every block time step.

Conventionally, graphics cards render into a "frame buffer", a special memory area that represents the image seen on a display. However, a frame buffer is unsuitable for our purposes as the data elements in this buffer are "clamped" to value ranges that map the capabilities of the display. Invariably this means that 32-bit real vectors are reduced in resolution and therefore in accuracy too. This is perfectly fine for visual displays where the number of colours after clamping are still 2^{24} (≈ 16 million), sufficient to make two neighbouring colours indiscernible to the human eye. However, this is unacceptable for scientific production calculations. The workaround is to create an off-screen frame buffer object and instruct GPU programs to render into these rather than to the screen. Off-screen frame buffers support 32-bit floating point values and are not clamped and therefore preserve their precision.

The GPU has two main kernel operations available in programmable graphics pipelines, these are a "vertex shader" and a "fragment shader". Our implementation only makes use of the fragment shader pipeline as it is better suited for the kind of calculations in the N-body problem and because the fragment pipeline in general provides more processing power[5]. The host CPU is responsible for allocating the input textures and frame buffer objects, copy the data between CPU and GPU, and binding textures that are to be processed by kernels. The lower order prediction and correction of the particle positions is done on the host CPU. In Tab. 1 we summarise the hardware properties of the two adopted GPU's and the GRAPE.

4 Results

To test the various implementations of the force evaluator we perform several tests on different hardware. For clarity we perform each test with the same

[5] Before the 8800GTX family of GPUs, vertex programs and fragment programs had to execute on distinct processing units. The 8800GTX is the first generation of GPUs where this distinction no longer exists and the two are unified.

Table 2
Results of the performance measurements for a Plummer sphere with N equal mass particles initially in virial equilibrium for $0.25N$-body time units (from $t = 0.25$ to $t = 0.5$) using a softening of $1/256$. In the first column we list the number of particles, followed by the timing results of the GRAPE in seconds. In the last column we give the timing results for the calculation without an attached processor. The GRAPE (second column) was measured up to 128k particles, because the on-board memory did not allow for larger simulations. Simulations on the FX1400 and the host computer were limited for practical reasons.

N	GRAPE-6Af	8800GTX	FX1400	Xeon
256	0.07098	2.708	3.423	0.1325
512	0.1410	8.777	10.59	0.5941
1024	0.3327	17.46	20.20	2.584
2048	0.7652	45.27	54.16	10.59
4096	1.991	128.3	157.8	50.40
8192	5.552	342.7	617.3	224.7
16384	16.32	924.4	3398	994.0
32768	51.68	1907	13180	4328
65536	178.2	3973	40560	19290
131072	-	8844	-	-
262144	-	22330	-	-
524288	-	63960	-	-

realization of the initial conditions. For this we vary the number of stars from $N = 256$ particles with steps of two to half a million stars (see Tab. 2). Not each set of initial condition is run on every processor, as the Intel Xeons, for example, would take a long time and the scaling with N is unlikely to change as we clearly have reached the CPU-limited calculation regime (see § 5).

The initial conditions for each set of simulations were generated by randomly selecting the stellar positions and velocities according to the Plummer (1911) distribution using the method described by Aarseth et al. (1974). Each of the stars were given the same mass. The initial particle representations were scaled to virial equilibrium before starting the calculation.

Each set of initial conditions is run from $t = 0$ to $t = 0.50$ time units (Heggie & Mathieu, 1986) [6], but the performance is measured only over the last quarter of an N-body time unit to reduce the overhead for reading the snapshot and the initialisation of the integrator. The maximum time step

[6] see also `http://en.wikipedia.org/wiki/Natural_units#N-body_units`

for the particles was 0.125, to guarantee that each particle was evaluated at least twice during the course of the simulation. The force calculations were performed by adopting a softening of 1/256 for all simulations.

For our performance measurements we have used four nodes of a Hewlett-Packard xw8200 workstation with a dual Intel Xeon CPU running at 3.4 GHz and either the GRAPE, a Quadro FX1400 or GeForce 8800GTX graphics card in the PCI Express (16×) bus. The cluster nodes were running a Linux SMP kernel version 2.6.16, Cg version 1.4, graphics card driver version 1.0-9746 and the OpenGL 2.0 bindings.

In Figure 1 we show the timing results of the N-body simulations. The FX1400 is slower than the general purpose computer over the entire range of N in our experiments. The bad performance of the FX1400 is mainly attributed to the additional overhead in communication and memory allocation. For $N \lesssim 10^4$ the GeForce 8800GTX GPU is slower than the host computer but continues to have a relatively flat scaling, comparable to the GRAPE-6, whereas the host has a much worse ($\propto N^2$) scaling. The scaling of the compute time of the GPU is proportional to that of the GRAPE ($\propto N^{3/2}$), but the latter has a smaller offset by about an order of magnitude. This is mainly caused by the efficient use of the GRAPE pipeline, which requires fewer clock cycles per force evaluation compared to the GPU (see § 6).

5 Performance modelling of the GPU

In modelling the performance of the GPU we adopt the model proposed by Makino (2002); Harfst et al. (2006) but tailored to the host plus GPU and to the GRAPE architecture.

The wall clock time required for advancing the $n_{\rm block}$ particles in a single block time step in the N-body systems is

$$t_{\rm step} = t_{\rm host} + t_{\rm force} + t_{\rm comm}. \tag{10}$$

Here $t_{\rm host} = t_{\rm pred} + t_{\rm corr}$ is the time spend on the host computer for predicting and correcting the particles in the block, $t_{\rm force}$ is the time spend on the attached processor and $t_{\rm comm}$ is the time spend communicating between the host and the attached processor. We now discuss the characteristics of each of the elements in the calculation for $t_{\rm step}$.

Host operation. The predictions and corrections of the particles are calculated on the host computer, and the time for this operation is directly related

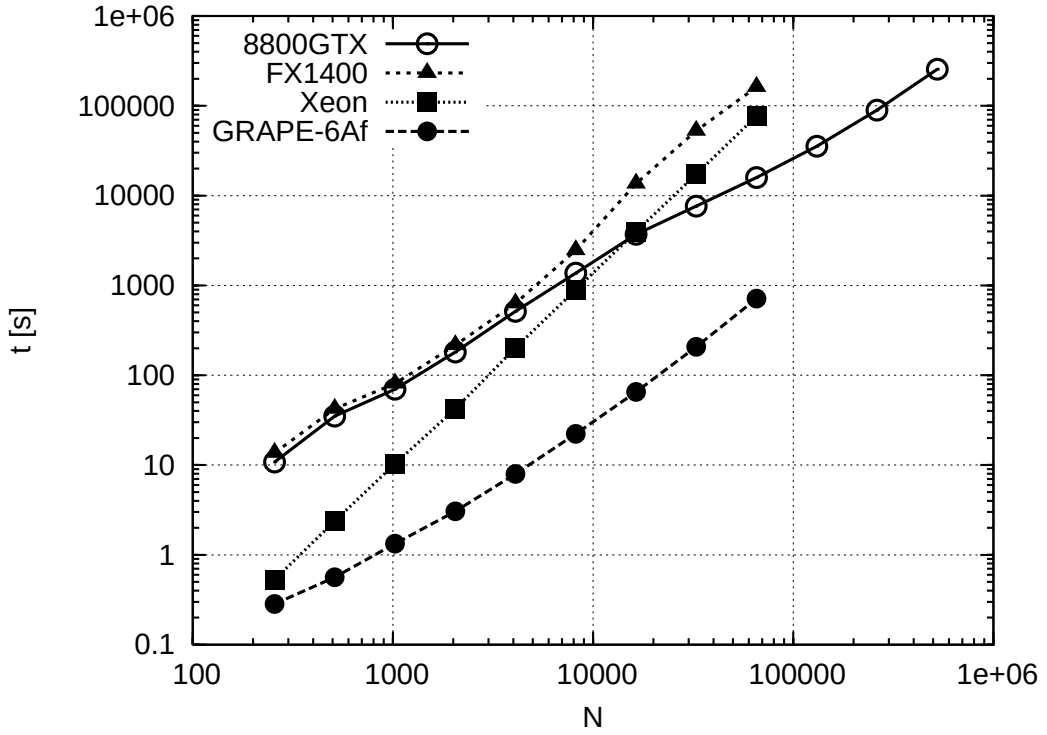

Fig. 1. Timing of several implementations of the gravitational N-body simulations for $N = 256$ particles to $N = 512$k particles (only for the 8800GTX, the others up to 64k) over one N-body time unit. The 8800GTX are represented with open circles connected with a solid curve, the GRAPE is given by bullets with dashed line. The thin dashed (triangles) line and thin dotted (squares) lines give the results of the calculations with the FX1400 and with only the host computer. Note that in timings in Tab. 2 were multiplied by a factor of four to estimate the compute time for one dynamical time unit, rather than the $1/4^{\rm th}$ over which the timing calculations were performed.

to the speed of the host processor $t_{\rm cpu}$, the number of operations in the prediction step $n_{\rm pred}$ and in the correction step $n_{\rm corr}$. The total time spend per block step then yields

$$t_{\rm pred} \simeq n_{\rm pred} t_{\rm cpu} N, \tag{11}$$

for the prediction and

$$t_{\rm corr} \simeq n_{\rm corr} t_{\rm cpu} N. \tag{12}$$

for the correction. The number of operations per prediction step $n_{\rm pred} \simeq 300$ and for the correction $n_{\rm corr} \simeq 1000$. This operation could be performed on the GPU, though the GRAPE is not designed for the predictior and corrector calculation. For a fair comparison between the GRAPE and the GPU and in order to preserve high accuracy we performed these calculations on the host.

Communication. The time spend communicating between the host and the attached processor is expressed by the sum of the time needed to send $n_{\rm send}$ particles to the acceleration hardware and the time needed to receive $n_{\rm rec}$ particles from the acceleration hardware:

$$t_{\rm comm} = \eta_{\rm send} t_{\rm send} n_{\rm send} + \eta_{\rm rec} t_{\rm rec} n_{\rm send}. \tag{13}$$

Here $\eta_{\rm send} t_{\rm send}$ and $\eta_{\rm rec} t_{\rm rec}$ are the time needed to send and receive one particle, respectively. For the computation without the hardware acceleration $t_{\rm comm} = 0$, since the forces between all particles are calculated locally. For the GRAPE and the GPUs, however, a considerable amount of time is spend in communication. For the GRAPE sending data is equally fast as receiving data, i.e: $t_{\rm send} = t_{\rm rec} = t_{\rm bus}$. Sending data to the GPU is considerably slower than receive data (see Tab. 3).

The two send and receive efficiency factors $\eta_{\rm send}$ and $\eta_{\rm rec}$, are the product of the overhead η_o and the number of bytes per particle that has to be send or received. The overhead $\eta_o = 188$ (Fukushige et al., 2005) for each of the attached processors. Since for the GRAPE the send and receive operation are equally expensive we can just count the number of bytes that has to be transported per particle, which for the GRAPE hardware is 72 bytes (Fukushige et al., 2005; Harfst et al., 2006). For the GRAPE we then write $\eta_{\rm send} t_{\rm send} + \eta_{\rm rec} t_{\rm rec} = 72 \times 188 t_{\rm bus}$.

For the GPU $\eta_{\rm send} > \eta_{\rm rec}$ (see Table 3 for the measured values). The additional overhead η_o is the same as for the GRAPE, but per particle the number of bytes to send is different from than the number to receive. As we discussed in § 3.1 a total of 56 bytes has to be send from the host to the GPU, whereas only 28 bytes are received. For the GPU we then write $\eta_{\rm send} = 56 \times 188$, whereas $\eta_{\rm rec} = 28 \times 188$.

In addition to the difference in the speeds for sending and receiving data, the GPU suffers from an additional penalty. The GRAPE sends the particles in the block ($n_{\rm send} = n_{\rm block}$), whereas due to internal memory management the GPU has to send and receive all particles $n_{\rm send} = N$ (see § 3.1). This efficiency loss is quite substantial, and will probably be reduced when we use CUDA as programming environment (see § 6).

For the adopted (Hermite predictor-corrector block time-step) integration scheme the number of particles in a single block $n_{\rm block}$ cannot be determined implicitly, though theoretical arguments suggest $n_{\rm block} \propto N^{2/3}$. Instead of using this estimate We fitted the average number of particles in a block time step. This fit was done with the equal mass Plummer sphere initial conditions running on GRAPE and run over one dynamical (N-body) time unit. The

average number of particles in a single block is then

$$n_{\rm block} \simeq 0.20 n^{0.81}. \tag{14}$$

Calculation. The time spend by the hardware acceleration ($t_{\rm force}$) is directly related to the speed of the dedicated processor ($t_{\rm GPU}$), the number of pipelines per processor ($n_{\rm pipe}$) and the number of operations for one force evaluation ($\eta_{\rm fe} \simeq 60$).

$$t_{\rm force} = \eta_{\rm fe} N n_{\rm block} t_{\rm GPU} / n_{\rm pipe}. \tag{15}$$

The details of the different hardware are presented in Tab. 1 and the measured values are in Tab. 3 The GRAPE has a vector pipeline for each processor which allows a more efficient force evaluation than the GPU's, the number of operations per force evaluation for the GRAPE therefore $\eta_{\rm fe} \simeq {\rm O}(1)$.

In order to enable hardware acceleration on our N-body code we had to introduce a number of additional operations, like reallocating arrays, which give rise to an extra computation overhead. For the calculations with the host computer without hardware acceleration we adopt $\eta_{\rm fe} \simeq 180$, a factor of three larger than for the GPUs.

Total performance. The total wall-clock time spent per dynamical (N-body) time unit is then

$$t = n_{\rm steps} t_{\rm step}. \tag{16}$$

Here we fitted to number of block steps per dynamical (N-body) time units. According to Makino & Hut (1988, 1990) $n_{\rm steps} \propto n^{1/3}$. We measured the number of block time steps using the equal mass Plummer distributions as initial conditions, using the GRAPE enabled code and fitted the result:

$$n_{\rm steps} \simeq 247 N^{0.35}. \tag{17}$$

In fig. 2 we compare the results of the performance model with the measurements on the workstation without additional hardware (squares) and with three attached processors; a single GRAPE-6Af processor board (bullets), an FX 1400 (triangles) and the newer GeForce 8800GTX (circles). Note that the measurements in Tab. 2 were multiplied by a factor four to compensate for the fact that we performed our timings only over a quarter N-body time unit. Though these curves are not fitted, they give a satisfactory comparison.

Table 3
Measurements for the various hardware using in this paper. The first line gives the time spent by the host computer for predicting and correcting a single particle. The second row is for calculating the force between two particles. The last two columns give the time to send a single particle to, and to receive a single particle from the attached hardware. For the calculations with only the host computer this operation is not available. In particular the communication with the GPUs turns out to be relatively slow.

param	GRAPE-6Af	8800GTX	FX 1400	Xeon
$t_{\rm host}$	3.82×10^{-7}	3.82×10^{-7}	3.82×10^{-7}	3.82×10^{-7}
$\eta_{\rm fe} t_{\rm force}$	1.11×10^{-8}	1.04×10^{-7}	1.72×10^{-7}	5.29×10^{-8}
$\eta_{\rm send} t_{\rm send}$	2.00×10^{-7}	1.76×10^{-5}	1.89×10^{-5}	NA
$\eta_{\rm rec} t_{\rm rec}$	2.00×10^{-7}	5.97×10^{-6}	5.98×10^{-6}	NA

The largest discrepancy between the performance model and the measurements can be noticed for the FX1400 GPU, which, for $N \gtrsim 10^4$ seems to perform considerably less efficiently than expected according to the performance model. Part of this discrepancy, though not explicitly mentioned in § 5 is in part the result of a hysteresis effect in the communication of both GPU's. For the 8800GTX, however, this effect is less evident, but still present. Both GPU's tend to have a maximum communication speed for blocks of 0.5 Mbyte (about 6000 particles). An additional effect which causing performance loss on the FX 1400 is the increase in the number of block time steps. This number continued to increase beyong our measurements performed with GRAPE (see Eq. 14).

The numbers listed in Tab. 3, and used in our performance model, are the optimum values. The communication speed drops by about a factor of two for much larger amounts of data transfer to and from the GPU. For the FX1400, this drop in communication is considerable, whereas for the 8800GTX it results in a smaller performance loss (mainly due to the larger number of processor pipelines). The discrepancy for the GRAPE calculation with low N is the result of neglecting the limited size of the processor pipeline in the performance model and due to the irregular behavior of the number of particles in each block time step.

In fig. 3 we present the speed-up for the various hardware configurations, compared to running on the host workstation. Here it is quite clear that for low N the GPU's do not give a appreciable speedup, but for a large number of particles, the GeForce 8800GTX gives a speedup of at least an order of magnitude, but not as much as the GRAPE. The latter, however, will not be able

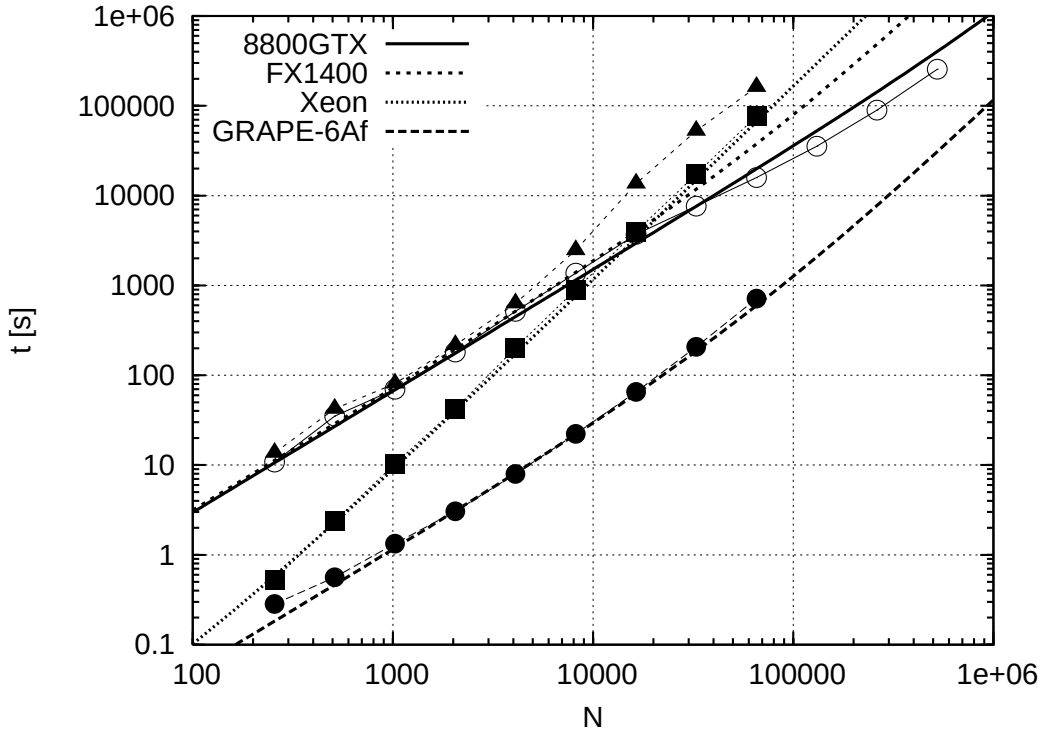

Fig. 2. The results of the above described performance model (thick lines) over-plotted with the results of the measurements for the three attached processors (symbols). The bullets represents the results from a single GRAPE-6Af processor board, the squares give the host workstation, the circles are for the GeForce 8800GTX and the triangles give the FX 1400 graphics processor.

to perform simulation of more than 128k particles [7] .

6 Discussion

We have successfully implemented the direct gravitational force evaluation calculation using Cg on two graphics cards, the NVIDIA Quadro FX1400 and the NVIDIA GeForce 8800GTX, and compared their performance with the host workstation and the GRAPE-6Af special purpose computer.

For $N \lesssim 10^4$ objects the workstation outperforms the GPUs. This is mainly due to additional overhead introduced by the communication to the GPU and memory allocation on the GPU. For a larger number of particles the more modern GPU (8800GTX) outperforms the workstation by up to about a factor of 50 (for 9 million particles). Such a large number of particles cannot

[7] Due to a defective chip on our GRAPE-6 the on-board memory was reduced from 128k particles to 64k particles. The latest GRAPE-6Af are equipped with 256k particles of memory.

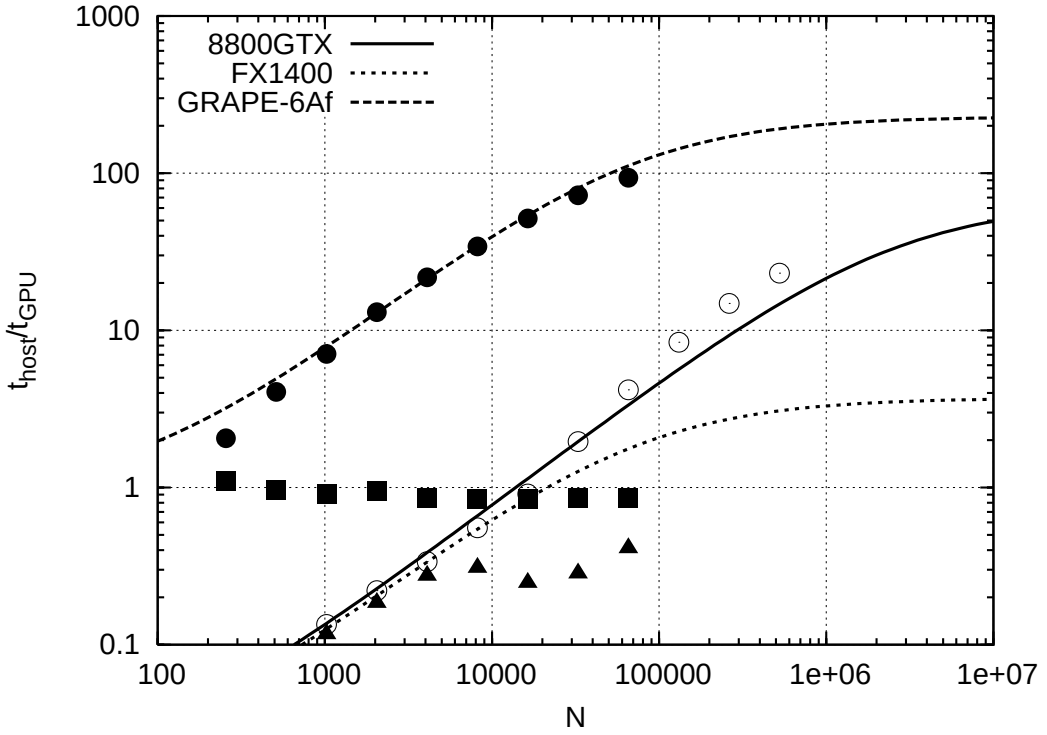

Fig. 3. The speedup of the two GPUs and the GRAPE with respect to the host workstation as a function of the number of particles. The (lower) dotted curve is for the Quadro FX1400, the solid curve (middle) gives the timing for the GeForce 8800GTX and the top line (dashes) represents the GRAPE.

be simulated on the GRAPE-6Af, due to memory limitations. For up to 256k, the maximum number of particles that can be stored on the GRAPE, the 8800GTX is slower than the GRAPE by about an order of magnitude. Still, at this particle number the GPU is faster than the workstation by an order of magnitude.

The GPU has lower accuracy compared to the GRAPE or the host workstation. The GRAPE-6 uses 24-bit mantissa for calculating the differential position, and 64bit fixed point format for accumulation. The pipeline for the time derivative is designed with 20bit mantissa and 32 bit fixed-point notation for the final accumulation (Makino et al., 2003). In principle the NVIDIA architecture should not be able to achieve similar precision, but would fall short in precision by about an order of magnitude compared to the GRAPE-6. The average mean error in the energy is $(1.7 \pm 1.6) \times 10^{-6}$ for the 8800GTX and $(5.1 \pm 0.56) \times 10^{-6}$ for the FX1400 (averaged over the simulations for $N = 256$ to $N = 64$k), whereas for the GRAPE we measured $(1.9 \pm 1.2) \times 10^{-7}$, which is comparable to the mean error on the host. Both the host and GRAPE produce an energy error which is about an order of magnitude smaller than that of the GPUs.

The precision of the GPU is regretfully unlikely to increase anytime soon, as the higher precision is not required for graphics applications and it would imply a considerable redesign of the hardware. But we could improve the accuracy of the GPU even further by sorting the forces on size before adding them, summing the smallest forces first.

The fixed point notation in the GRAPE-6 allows for a much more efficient use of clock cycles, allowing effectively one operation per clock cycle, whereas the NVIDIA architecture requires more cycles. This turns out to be an important reason why the 8800GTX is slower than the GRAPE-6.

The main advantage of the GPU over that of the dedicated GRAPE hardware, is the much larger memory, the wider applicability and the much lower cost of the former. The large memory on the GPU allows simulations of up to about 9 million particles, though one has to wait for about two years for one dynamical time scale.

In theory the 8800GTX should be able to outperform the GRAPE-6Af, but due to relatively inefficient memory access and additional overhead cost, which is not present in the GRAPE hardware, many clock cycles seem to be lost. With a more efficient use of the hardware the GPU could, in principle, improve performance by about two orders of magnitude. For the next generation of GPUs we hope that this efficiency bottleneck will be lifted. In that case, the GPU would outperform GRAPE by almost an order of magnitude. Note, however, that the GRAPE-6 is based on 5 year old technology, and the next generation GRAPE is likely to outperform modern GPUs by a sizable margin.

These current bottlenecks in the GPU may be reduced using the compute unified device architecture (CUDA) [8] programming environment, which is supposed to provide an improved environment for general purpose programming on the GPU. In fig. 4 we present the possible future performance assuming that the communication additional overhead on the GPU is lifted, the clock cycles are used more efficiently without any assumptions of improved hardware speed. In the first step we simple reduce communication to blocks rather than having to transport all particles each block time step (solid curve). This relatively simple improvement can already be carried out using CUDA. The second optimalization (dashed curve in fig. 4) is achieved when, in addition to reducing the communication we also carried the predictor and corrector steps to the GPU. This improvement, however, may be associated with a quite severe accuracy penalty. For both improvements we used the performance data for the current design 8800GTX. Further improvement can be achieved when, in addition to more efficient communication the force pipeline can be represented more efficiently by the shader pipeline, like is done on GRAPE. The

[8] see `http://developer.nvidia.com/cuda`

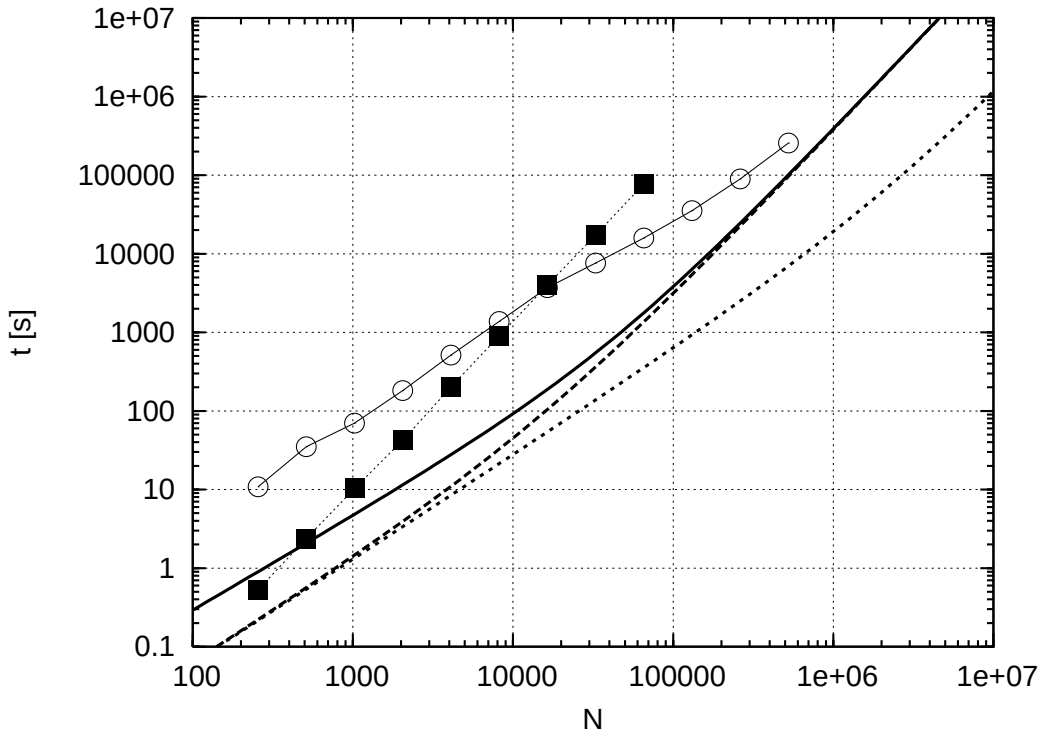

Fig. 4. Prospective of future CPU and GPU performance, based on the model from § 5. The two thin curves with squares and circles give the measured performance of the CPU and 8800GTX GPU, respectively. The thick solid curve gives a prediction of the performance for the 8800GTX in which only blocks of particles are communicated with the GPU. The dashed curves gives in addition the effect of carrying the predictor/corrector calculation to the GPU. The doted curve gives the performance of a hypothetical 8800GTX-like architecture for which in addition the processor pipeline would be used more efficiently ($\eta_{\rm fe} = 1$).

result of this hypothetical case would improve performance by more than a factor 100 compared to the workstation over the entire range of N.

Acknowlegments

We are grateful to Mark Harris and David Luebke of NVIDIA for supplying us with the two NVIDIA GeForce 8800GTX graphics cards on which part of the simulations were performed. We also like to thanks Alessia Gualandris and Jun Makino for numerous discussions. This work was supported by NWO (via grant #635.000.303 and #643.200.503) and the Netherlands Advanced School for Astrophysics (NOVA). The calculations for this work were done on the Hewlett-Packard xw8200 workstation cluster and the MoDeStA computer in Amsterdam, both are hosted by SARA computing and networking services, Amsterdam.

References

Aarseth, S. J. 1999, PASP , 111, 1333

Aarseth, S. J., Henon, M., Wielen, R. 1974, A&A , 37, 183

Aarseth, S. J., Hoyle, F. 1964, Astrophysica Norvegica, 9, 313

Aarseth, S. J., Lecar, M. 1975, ARA&A , 13, 1

Applegate, J. H., Douglas, M. R., Gürsel, Y., Hunter, P., Seitz, C. L., Sussman, G. J. 1986, in P. Hut, S. L. W. McMillan (eds.), LNP Vol. 267: The Use of Supercomputers in Stellar Dynamics, p. 86

Buck, I., Foley, T., Horn, D., Sugerman, J., Mike, K., Pat, H. 2004, Brook for GPUs: Stream Computing on Graphics Hardware

Dorband, E. N., Hemsendorf, M., Merritt, D. 2003, Journal of Computational Physics, 185, 484

Elsen, E., Houston, M., Vishal, V., Darve, E., Hanrahan, P., Pand, V., 2006, To appear in "SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing", ACM Press, New York.

Göddeke, D., 2005, GPGPU–Basic Math Tutorial, *Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 300*

Fernando, R. 2004, GPU Gems (Programming Techniques, Tips, and Tricks for Real-Time Graphics), Addison Wesley, ISBN 0-321-22832-4

Fernando, R., Kilgard, M. J. 2003, The Cg Tutorial (The Definitive Guide to Programmable Real-Time Graphics), Addison Wesley, ISBN 0-321-19496-9

Fukushige, T., Makino, J., Kawai, A. 2005, Publ. Astr. Soc. Japan , 57, 1009

Gualandris, A., Portegies Zwart, S., Tirado-Ramos, A. 2007, PARCO in press, ArXiv Astrophysics e-prints (astro-ph/0608125)

Harfst, S., Gualandris, A., Merritt, D., Spurzem, R., Portegies Zwart, S., Berczik, P. 2007, New Astronomy in press, ArXiv Astrophysics e-prints (astro-ph/0608125)

Heggie, D. C., Mathieu, R. D. 1986, in P. Hut, S. L. W. McMillan (eds.), LNP Vol. 267: The Use of Supercomputers in Stellar Dynamics, p. 233

Hoekstra, A., Portegies Zwart, S., Bubak, M., Sloot, P., 2007, submitted to CRC Press LLC, ArXiv Astrophysics e-prints (astro-ph/07XXXXX)

Holmberg, E. 1941, ApJ , 94, 385

Hut, P. 2007, presented at A Life With Stars (Conference in Honor of Ed van den Heuvel), Amsterdam, August, 2007, ArXiv Astrophysics e-prints (astro-ph/0601232)

Makino, J. 1991, ApJ , 369, 200

Makino, J. 2001, in S. Deiters, B. Fuchs, A. Just, R. Spurzem, R. Wielen (eds.), ASP Conf. Ser. 228: Dynamics of Star Clusters and the Milky Way, p. 87

Makino, J. 2002, New Astronomy, 7, 373

Makino, J. 2005a, Journal of Korean Astronomical Society, 38, 165

Makino, J. 2007, ArXiv Astrophysics e-prints (astro-ph/0509278)

Makino, J., Aarseth, S. J. 1992, Publ. Astr. Soc. Japan , 44, 141

Makino, J., Fukushige, T., Koga, M., Namura, K. 2003, Publ. Astr. Soc.

Japan , 55, 1163
Makino, J., Hut, P. 1988, ApJS , 68, 833
Makino, J., Hut, P. 1990, ApJ , 365, 208
Makino, J., Taiji, M. 1998, Scientific simulations with special-purpose computers : The GRAPE systems, Scientific simulations with special-purpose computers : The GRAPE systems /by Junichiro Makino & Makoto Taiji. Chichester ; Toronto : John Wiley & Sons, c1998.
McMillan, S. L. W., Aarseth, S. J. 1993, ApJ , 414, 200
Moore, G. E. 1965, Electronics, 38(8)
Nitadori K., Makino J., Hut P., 2006, NewA, 12, 169
Nitadori K., Makino J., Abe G., 2007, To appear in the conference proceedings of Computational Science 2006, ArXiv Astrophysics e-prints (astro-ph/0606105)
Nyland, L., Harris, M., Prins, J., 2004, Poster presented at *The ACM Workshop on General Purpose Computing on Graphics Hardware*, Aug. 7-8, Los Angeles, CA.
Pharr, M., Fernando, R. 2005, GPU Gems 2 (Programming Techniques for High-Performance Graphics and General-Purpose Computation), Addison Wesley, ISBN 0-321-33559-7
Plummer, H. C. 1911, MNRAS , 71, 460
Taiji, M., Makino, J., Fukushige, T., Ebisuzaki, T., Sugimoto, D. 1996, in P. Hut, J. Makino (eds.), IAU Symp. 174: Dynamical Evolution of Star Clusters: Confrontation of Theory and Observations, p. 141
van Albada, T. S. 1968, BAN , 19, 479
von Hoerner, S. 1963, Zeitschrift fur Astrophysik, 57, 47

Appendix A

The N-body code presented in this paper consists of a part implemented in C (running on a CPU) and a part implemented in Cg (running on the GPU). In this appendix we show the routine that evaluates the acceleration, jerk and potential in Cg (which was based on a tutorial available from Göddeke (2005)). The C code which handles communication between CPU and GPU and supporting data structures is not presented here. A copy of the entire working version of the code is available via `http://modesta.science.uva.nl`.

```
void compute_acc_jerk_and_pot(
  in  float2 coords     : TEXCOORD0,      // 2D texture coordinate of this particle
  out float3 acc        : COLOR0,         // Output texture with acceleration
  out float4 jerkAndPot : COLOR1,         // Output texture with jerk and potential
  uniform samplerRECT accTexture,         // Input texture with all particles' acceleration
  uniform samplerRECT jerkAndPotTexture,  //  ,,     ,,     ,,   ,,    ,,      jerk and potential
  uniform samplerRECT massTexture,        //  ,,     ,,     ,,   ,,    ,,      mass
  uniform samplerRECT posTexture,         //  ,,     ,,     ,,   ,,    ,,      position
  uniform samplerRECT velTexture,         //  ,,     ,,     ,,   ,,    ,,      velocity
  uniform float eps2,                     // Softening parameter
  uniform float otherParticle,            // Index to other particle
  uniform float texSizeX,                 // Width of all textures
  uniform float texSizeY,                 // Height of all textures
  uniform float offset)                   // Number of unused texture elements
{
  float  coords1D, newCoords1D, otherMass,
         r2, xdotv, r2inv, rinv, r3inv, r5inv, xdotvr5inv;
  float2 newCoords;
  float3 pos, otherPos, vel, otherVel, dx, dv, thisAcc, thisJerkAndPot;

  // Get data from the textures
  acc        = texRECT(accTexture, coords).rgb;
  jerkAndPot = texRECT(jerkAndPotTexture, coords).rgba;
  pos        = texRECT(posTexture, coords).rgb;
  vel        = texRECT(velTexture, coords).v7.texrgb;

  // Convert the 2D texture coordinate to 1D and increase with otherParticle
  // to obtain the coordinate of this iteration's other particle. Because our
  // textures are defined as samplerRECT, texture elements must be addressed
  // as (x+0.5,y+0.5). When converting to 1D, we must compensate for this offset).
  coords1D = round(coords.y-0.5)*texSizeX + round(coords.x-0.5);
  newCoords1D = coords1D + otherParticle;

  // Skip over unused texture elements
  if (newCoords1D + offset > texSizeX*texSizeY - 1)
    newCoords1D = newCoords1D - (texSizeX*texSizeY - offset);

  // Convert the other particle's 1D coordinate to 2D. As above, we must add
  // 0.5 to obtain correct texture element coordinates.
  newCoords = 0.5 + float2( frac(newCoords1D/texSizeX)*texSizeX,
                            floor(newCoords1D/texSizeX) );

  // Get the position, velocity and mass of this iteration's other particle
  otherPos  = texRECT(posTexture, newCoords).rgb;
  otherVel  = texRECT(velTexture, newCoords).rgb;
  otherMass = texRECT(massTexture, newCoords).r;

  // Compute acceleration, jerk and potential
  dx = otherPos-pos;
  dv = otherVel-vel;
  r2 = eps2 + dot(dx,dx);
  xdotv = dot(dx,dv);
  r2inv = 1.0/r2;
  rinv = sqrt(r2inv);
  r3inv = r2inv*rinv;
  r5inv = r2inv*r3inv;
  xdotvr5inv = 3.0*xdotv*r5inv;
  thisAcc = otherMass*r3inv*dx;
  thisJerkAndPot = otherMass*(r3inv*dv - xdotvr5inv * dx);
  acc = acc + thisAcc;
  jerkAndPot.rgb = jerkAndPot.rgb + thisJerkAndPot;
  jerkAndPot.a = jerkAndPot.a - otherMass*rinv;
}
```